Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 7: this Friday @ 11:59 pm
  - Lab 6: next Monday 10/31 @ 11:59 pm
  - **Assignment 2: Friday 11/4 @ 11:59 pm**
  - Lab 7: Monday 11/7 @ 11:59 pm
- Live Support Session for Assignment 2
  - This Friday 7-8 pm (https://pitt.zoom.us/my/khattab)
- Weekly Live QA Session on Piazza
  - Friday 4:30-5:30 pm

Previous lecture

- ADT Graph
  - definitions
  - representations
    - two-arrays
    - adjacency matrix
    - adjacency lists
  - traversals
    - BFS
      - shortest paths based on number of edges
      - connected components

This Lecture

- ADT Graph
  - traversals
    - DFS
      - finding articulation points of a graph
  - representation
    - Graph compression

Problem of previous lecture

- **Input**: A file containing LinkedIn (LI) accounts and their connections
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …

- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - e.g., 1st connection?, 2nd connection?, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - For each connected component, are there certain accounts that if removed, the remaining accounts become ***partitioned***?

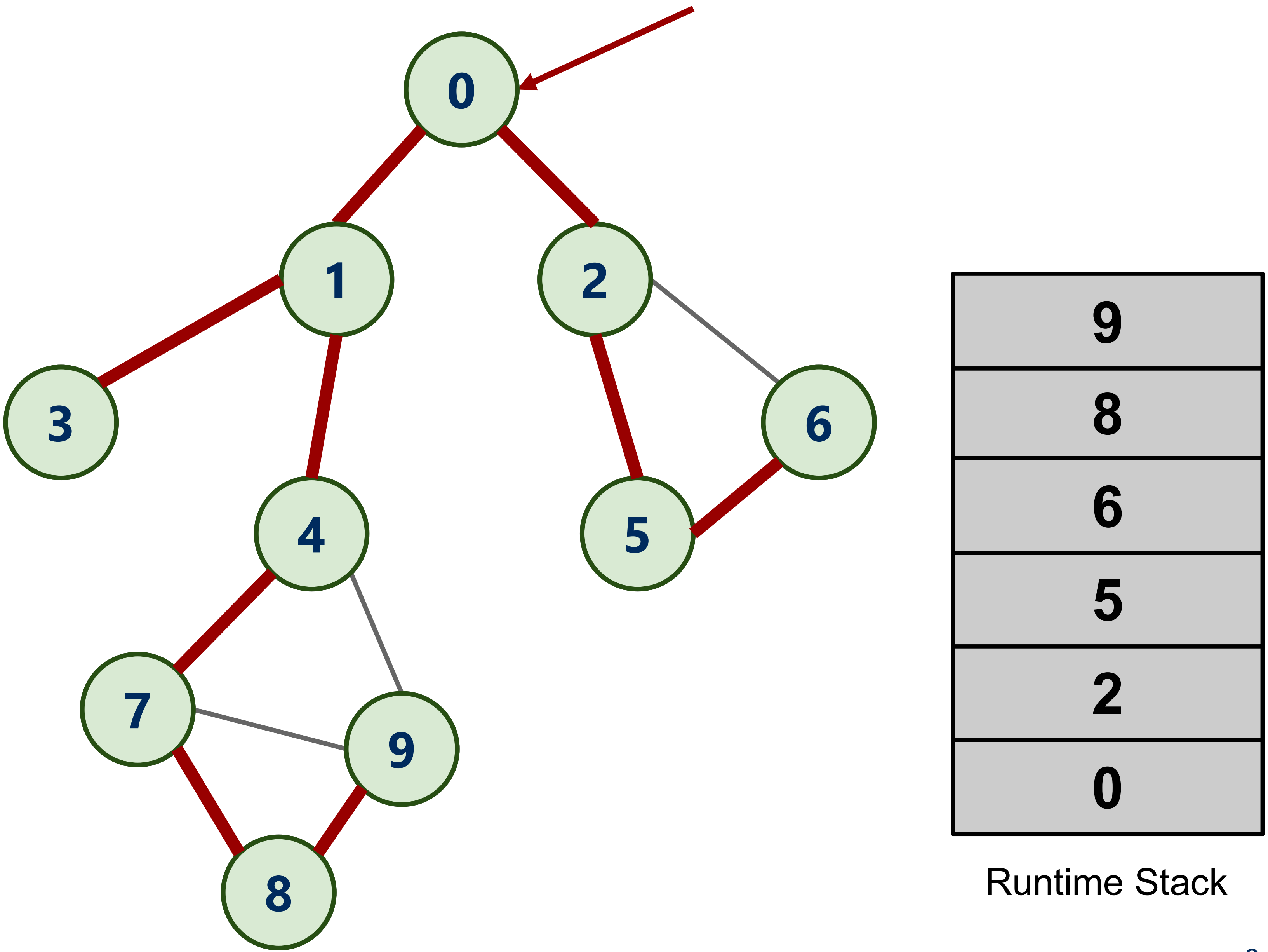
DFS – Depth First Search

- Already seen and used this throughout the term
  - For Huffman encoding…
    - as we build the codebook out of the Huffman Trie
- Can be easily implemented recursively
  - For each vertex, visit *first* unseen neighbor
  - Backtrack at deadends (i.e., vertices with no unseen neighbors)
    - Try *next* unseen neighbor after backtracking
  - An arbitrary order of neighbors is assumed

DFS Pseudo-code

```
DFS(vertex v) {
  seen[v] = true //mark v as seen
  for each unseen neighbor w
    parent[w] = v
    DFS(w)
}
```

DFS example



When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
   visit v //pre-order DFS
   for each unseen neighbor w
      parent[w] = v
      DFS(w)
}
```

When to visit a vertex

```
DFS(vertex v) {
  seen[v] = true //mark v as seen
for each unseen neighbor w
    parent[w] = v
    DFS(w)
 visit v //post-order DFS
}
```

When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
for each unseen neighbor w
      parent[w] = v
      DFS(w)
      (re)visit v //in-order DFS
}
```

Runtime Analysis of BFS

- Each vertex is added to the queue exactly once and removed exactly once
  - *v* add/remove operations
    - *O(v)* time for vertex processing
- Edges are checked when adding the list of neighbors to the queue
- Each edge is checked at most twice, one per edge endpoint
  - *O(e)* time for edge processing
- Total time: vertex processing time + edge processing time
  - *O(v + e)*

Runtime Analysis for DFS

- For Adjacency Matrix representation, BFS checks each *possible* edge!
  - $O(v^2)$ time for edge processing with Adjacency Matrix
- Total time: $O(v^2 + v) = O(v^2)$

Runtime Analysis of DFS

- Each vertex is seen then visited exactly once
  - *O(v)* time for vertex processing
  - except when (re)visiting a vertex after each child
    - vertex processing happens inside edge processing in that case
- Edges are checked when finding the list of neighbors
- Each edge is checked at most twice, one per edge endpoint
  - *O(e)* time for edge processing
- Total time: vertex processing time + edge processing time
  - *O(v + e)*

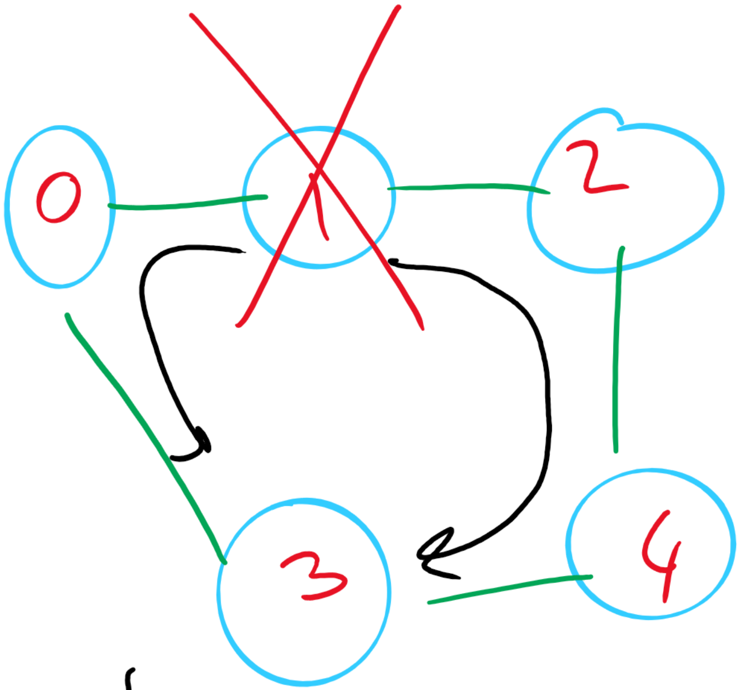
Runtime Analysis of BFS and DFS

- At a high level, DFS and BFS have the same runtime
  - Each vertex must be seen and then visited, but the order will differ between these two approaches
- The representation of the graph affect the runtimes of of these traversal algorithms?
  - $O(v + e)$ with Adjacency Lists
  - $O(v^2)$ with Adjacency Matrix
  - Note that for a dense graph, $v + e = O(v^2)$

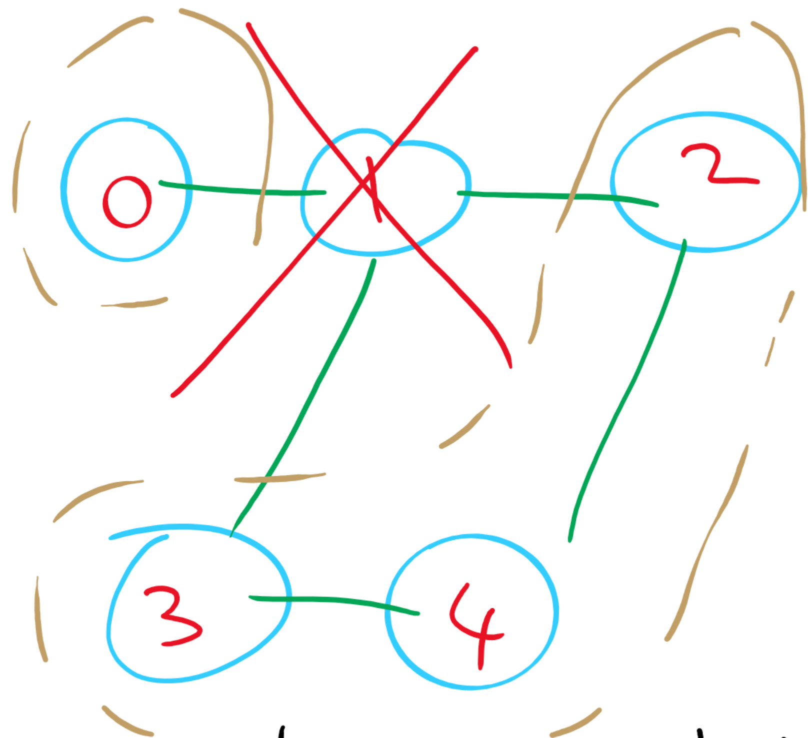
Biconnected graphs

- A *biconnected graph* has at least 2 distinct paths between all vertex pairs
  - a distinct path shares no common edges or vertices with another path except for the start and end vertices
- A graph is biconnected graph iff it has zero *articulation points*
  - Vertices, that, if removed, will separate the graph

Biconnected Graph



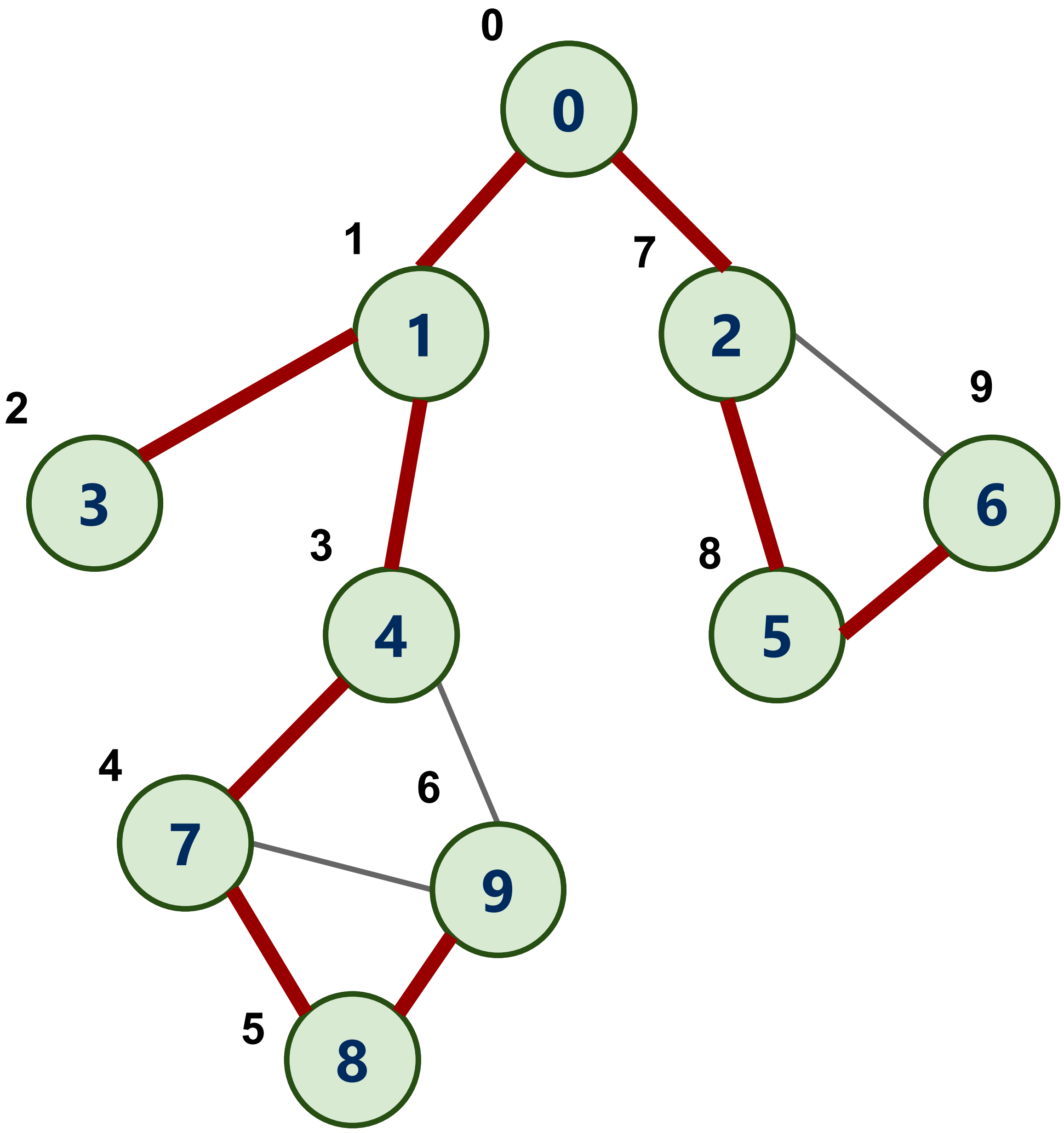
bi-connected



not bi-connected

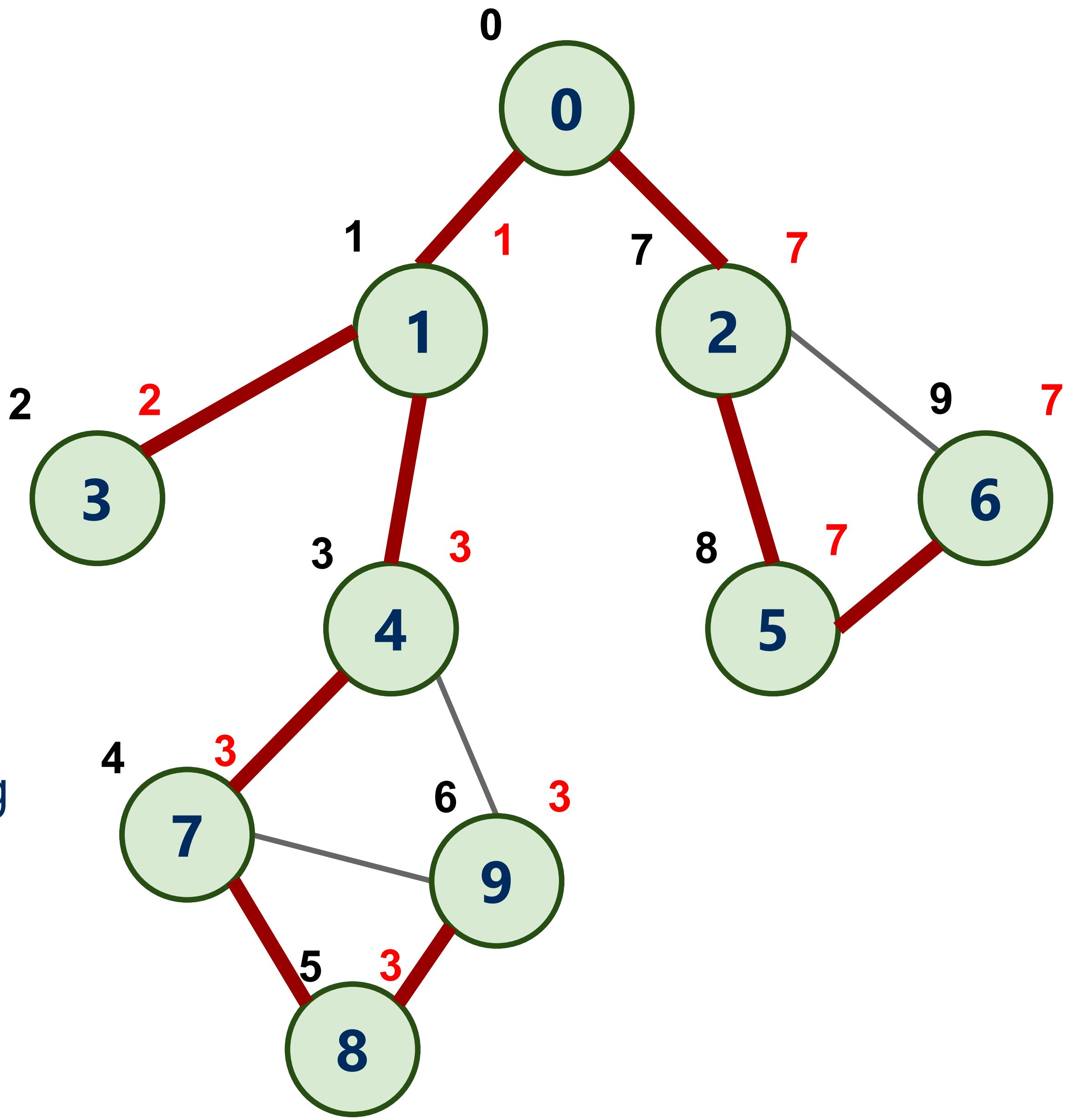
Finding articulation points of a graph

- Edges not included in the spanning tree are called **back edges**
  - e.g., (4, 9) and (2, 6)
- A pre-order DFS traversal visits the vertices in some order
  - let's number the vertices with their traversal order
  - num(v)



Finding articulation points of a graph

- For each non-root vertex v, find the lowest numbered vertex reachable from v
  - not through v's parent
  - using 0 or more tree edges then at most one back edge
- move down the tree looking for a back edge that goes backwards the furtheset

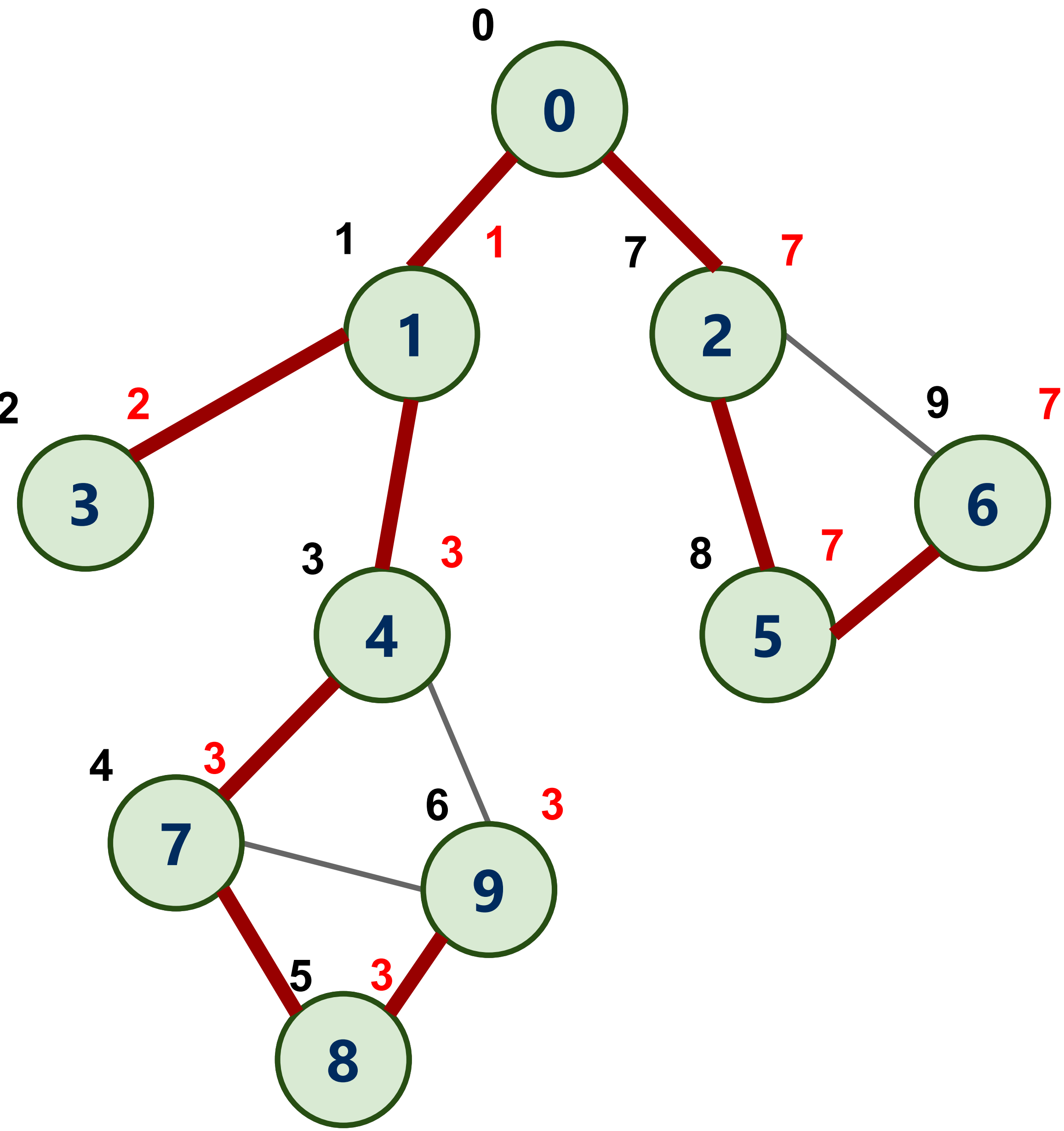


low(v)

- low(v) = lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then **at most one** back edge
  - Min of:
    - num(v) (the vertex is reachable from itself)
    - Lowest num(w) of all back edges (v, w)
    - Lowest low(w) of all children of *v* (the lowest-numbered vertex reachable through a child)

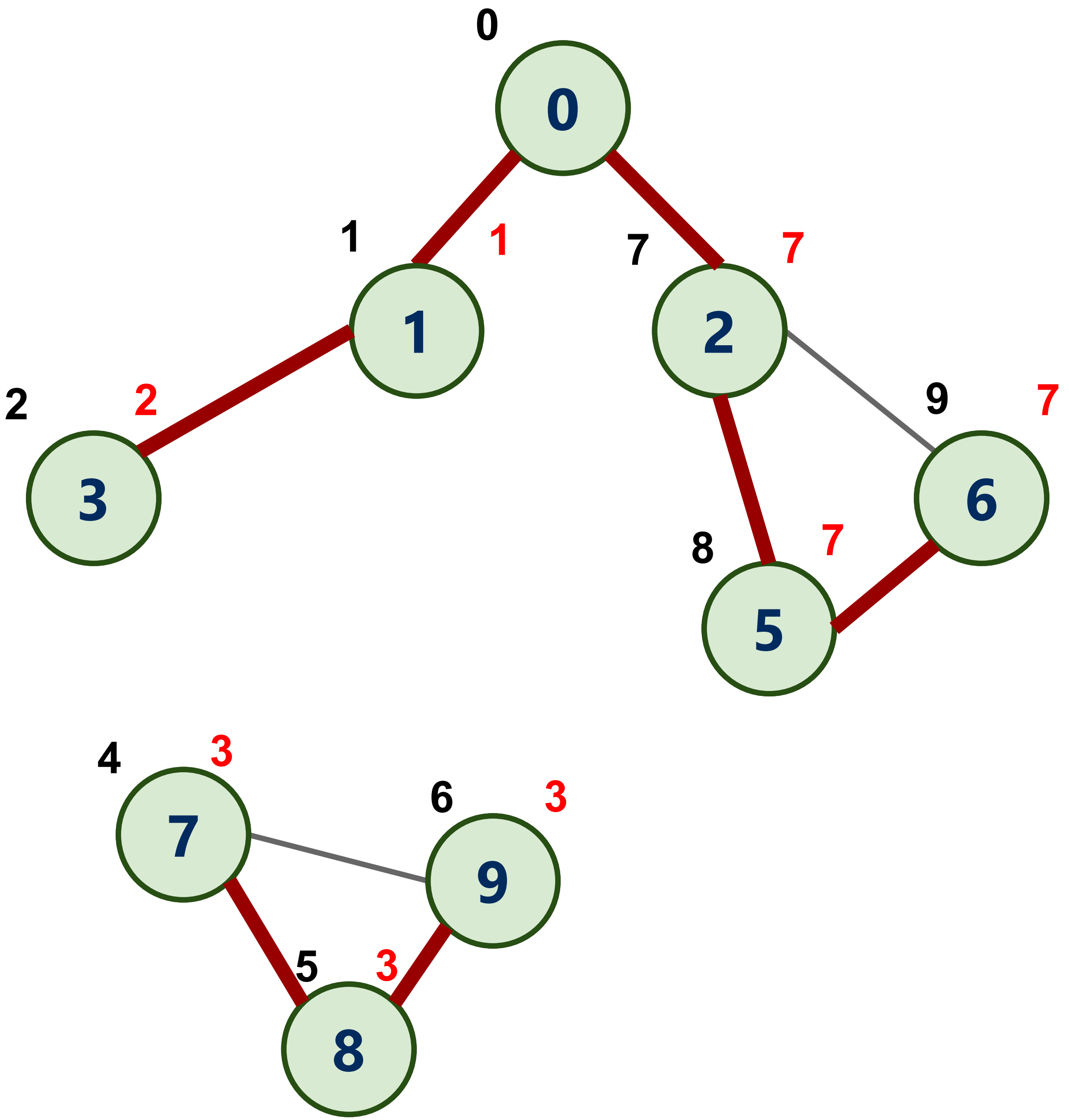
Finding articulation points of a graph

- What does it mean if a vertex v has a child w such that
  - low(w) >= num(v)?
- e.g., 4 and 7
- It means the child has no other way except through its parent to reach at least one other vertex
- e.g., 7 cannot reach 0, 1, and 3 except through 4
- So, the parent is an articulation point!



Finding articulation points of a graph

- So, the parent is an articulation point!
  - e.g., if 4 is removed, the graph becomes disconnected
- Each non-root vertex v that has a child w with low(w) >= num(v) is an articulation point



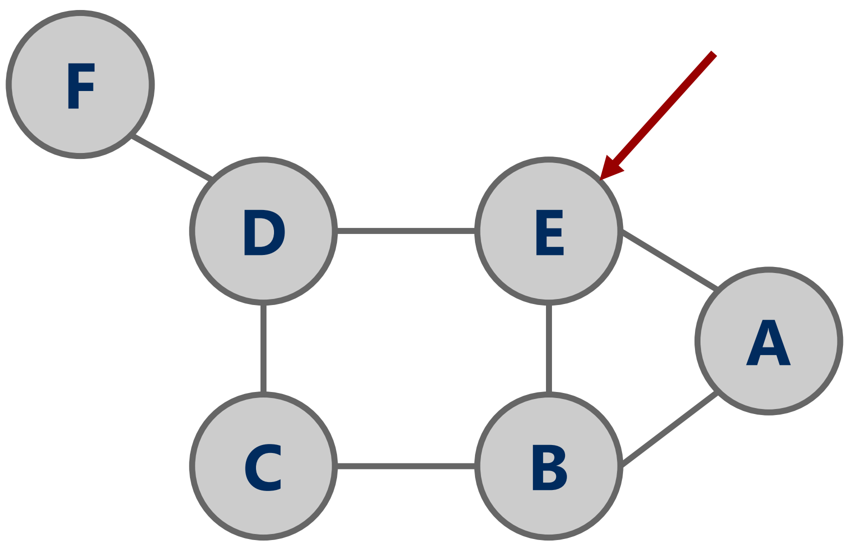
Finding articulation points of a graph: The Algorithm

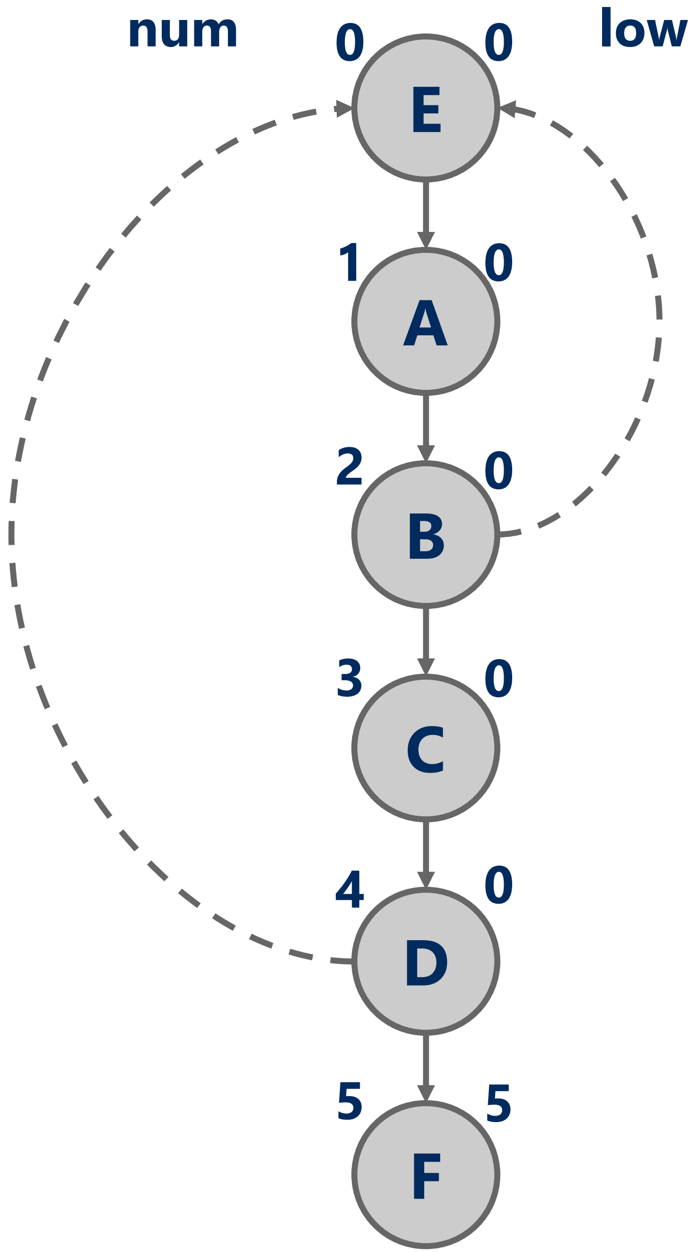
- Build a DFS spanning tree
  - Think of it as directed
  - Create back edges when considering a vertex that has already been visited
  - Label each vertex v with with two numbers:
    - num(v) = pre-order traversal order
    - low(v) = lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then at most one back edge

when to compute num(v) and low(v)

- num(v) is computed as we move down the tree
  - pre-order DFS
- low(v) is computed as we move up the tree
- Recursive DFS is convenient to compute both
  - why?

Finding articulation points example





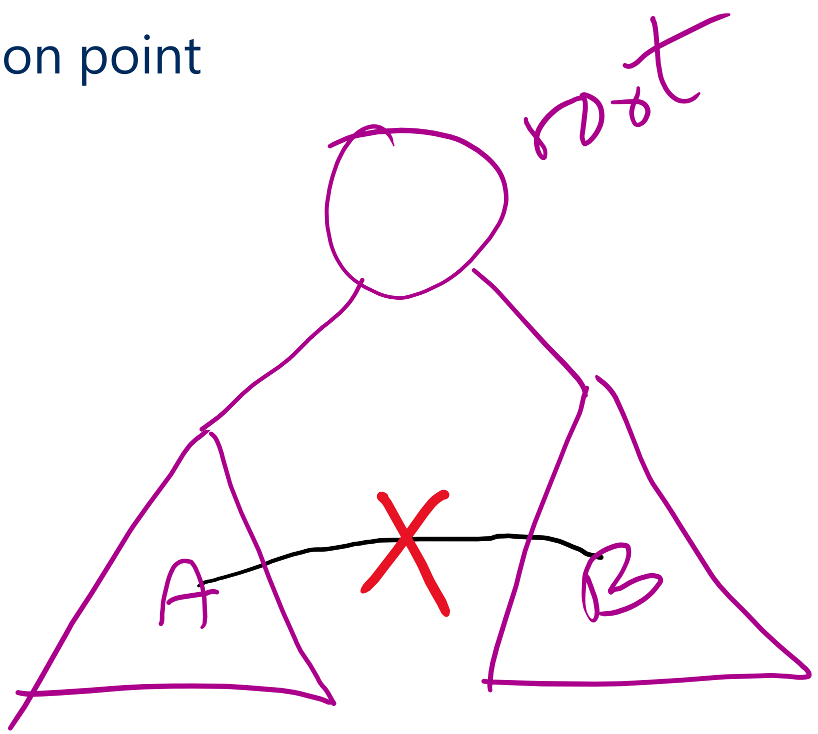
Using DFS to find the articulation points of a connected undirected graph

```
int num = 0

DFS(vertex v) {
      num[v] = num++
      low[v] = num[v] //initially
      seen[v] = true //mark v as seen
      for each neighbor w
         if(w unseen){
            parent[w] = v
            DFS(w) //after the call returns low[w] is computed, why?
            low[v] = min(low[v], low[w])
         } else { //seen neighbor
            if(w != parent[v]) //and not the parent, so back edge
              low[v] = min(low[v], num[w])
         }
}
```

What about the root of the spanning tree?

- What if we start DFS at an articulation point?
  - The starting vertex becomes the root of the spanning tree
  - If the root of the spanning tree has more than one child, the root is an articulation point

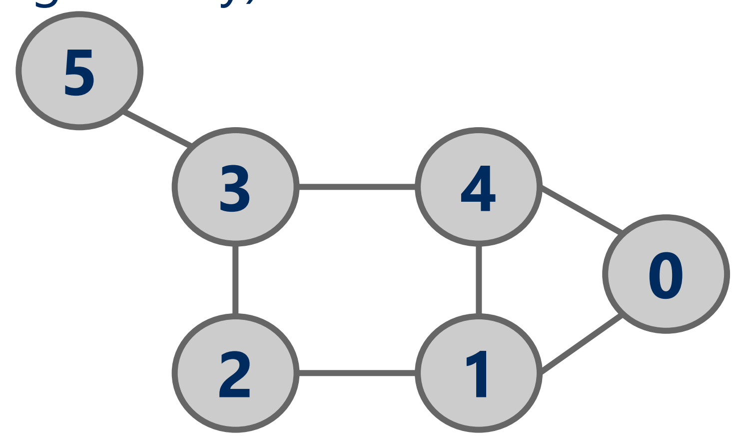


Graph Compression

- Real-life graphs are huge
  - 100’s if not 1000’s of GBs
  - Facebook graph, Google graph, maps, …
- Let’s see one (partial) idea for reducing the size of large graphs

Graph Compression

- **Step 1:** Construct a Compressed Sparse Row (CSR) representation of the graph
- CSR
  - Edges array concatenates *sorted* neighbor lists of all vertices
  - Offsets array:
    - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v

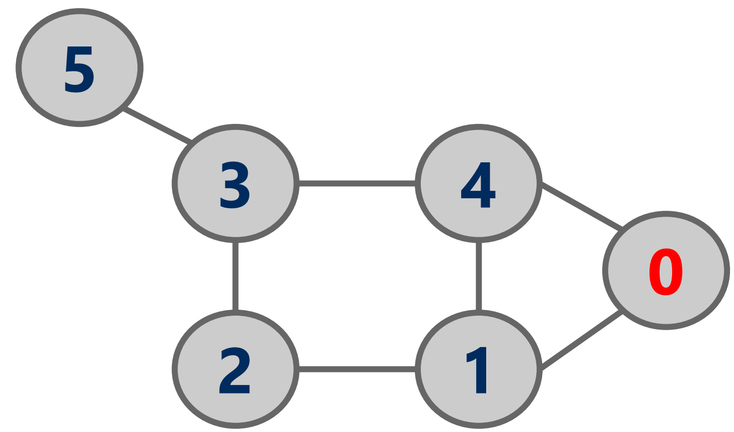


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets	0	2	5	7	10	13

Graph Compression

- Let’s start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

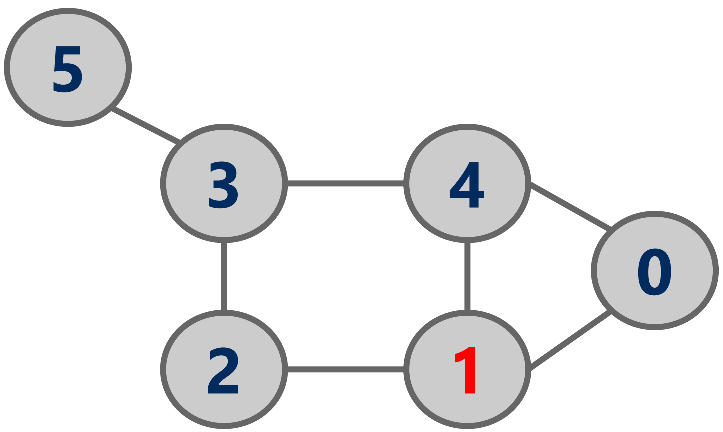
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

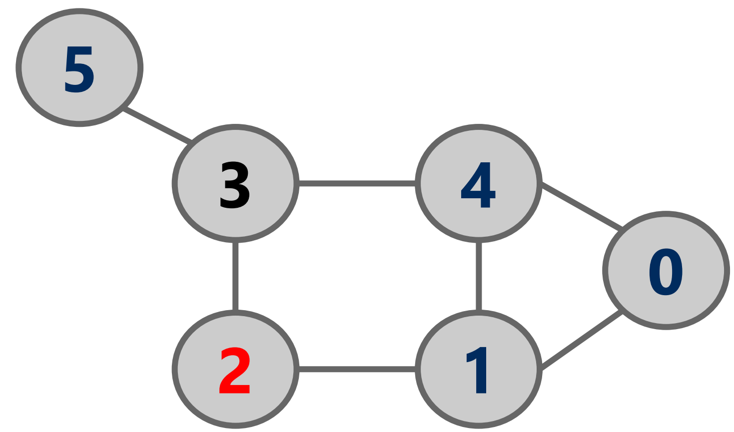
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

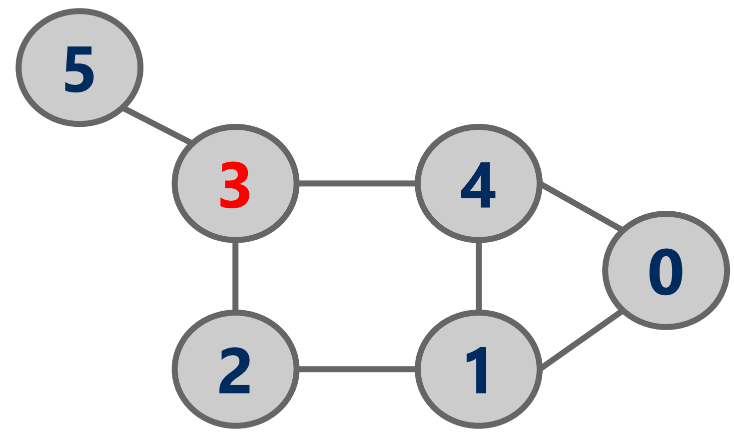
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let’s start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

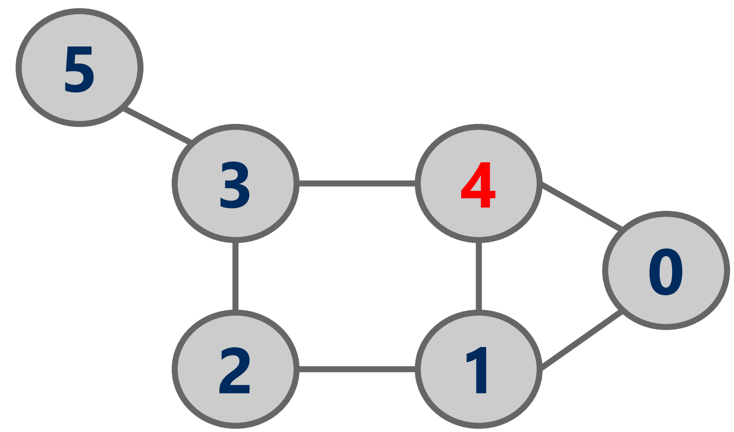
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let’s start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

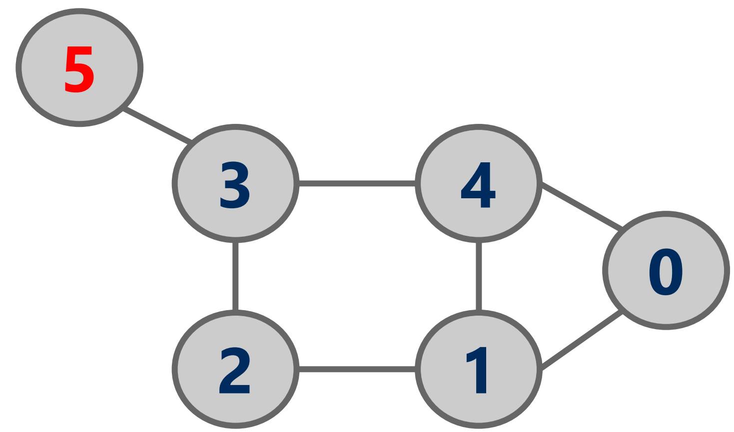
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

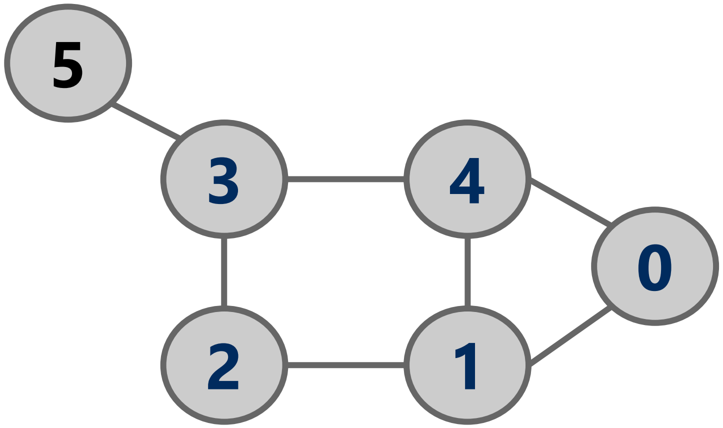
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Can we compute the degree of a vertex using the offsets array?
  - Running time?
- What is the required space of this representation?
  - Theta(m + n)
  - Assume 4 bytes per vertex and per edge
  - Total size: *4*v* + *8*e* bytes



Edges

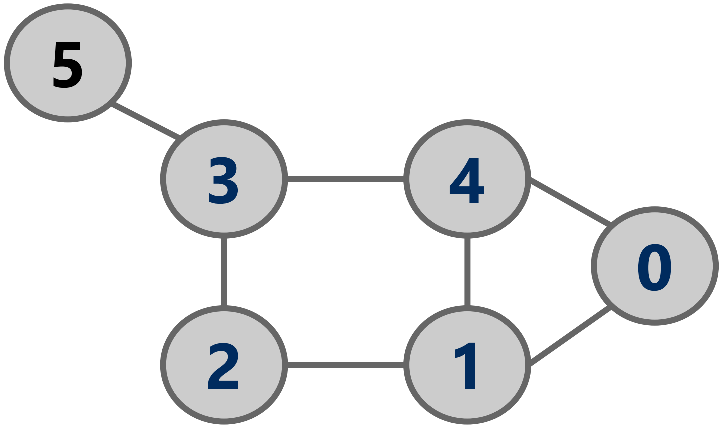
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list $v_1, v_2, v_3, \ldots$
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

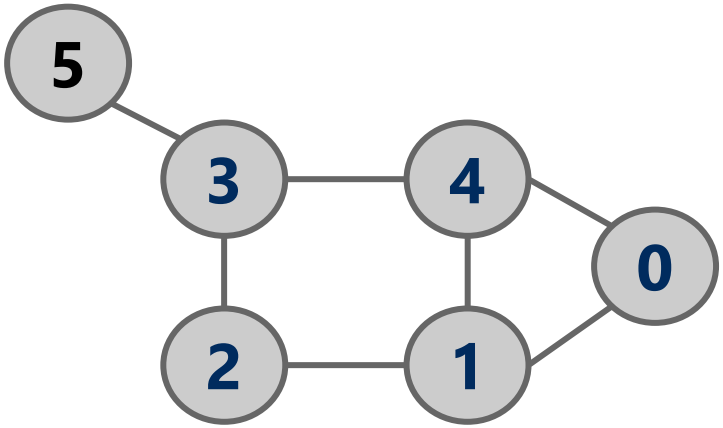
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

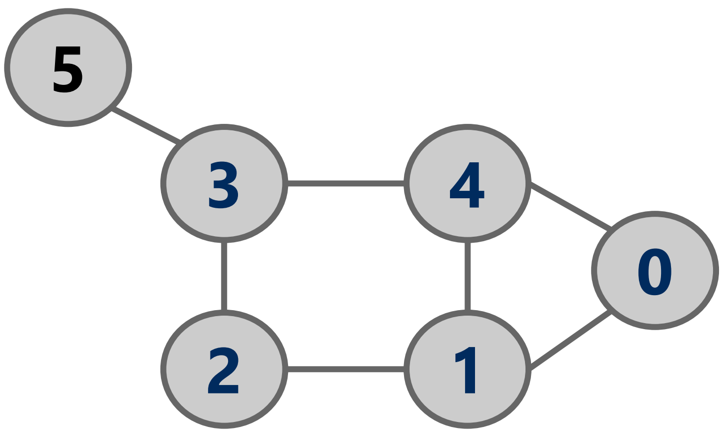
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1-0	4-1												

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list $v_1, v_2, v_3, \ldots$
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

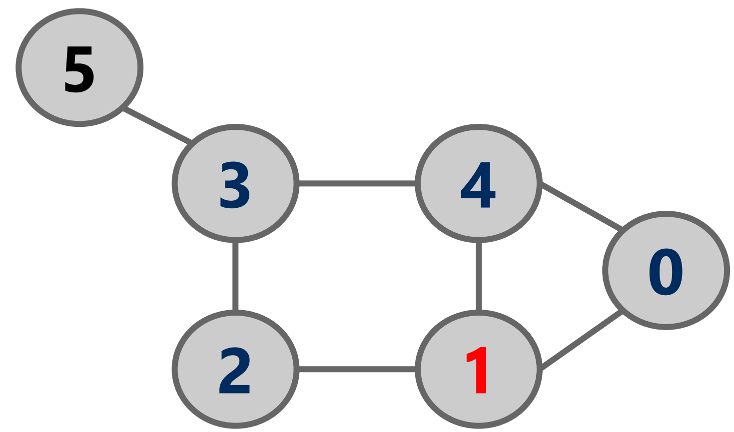
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	4	0	2	4	1	3	2	4	5	0	1	3	3
	1	3												

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$

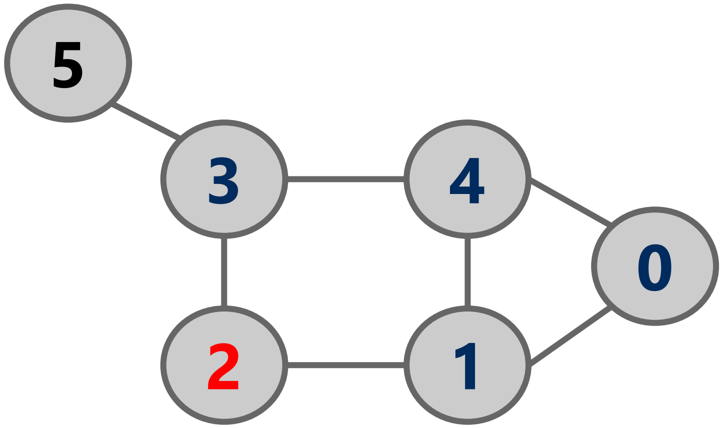


	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3
	1	3	-1	2	2									

Offsets	0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

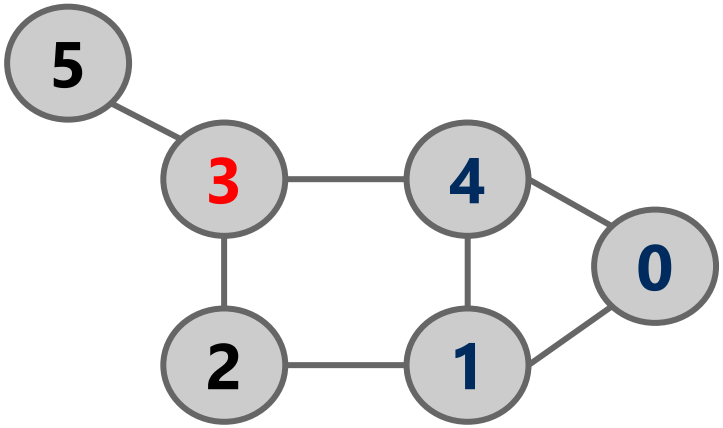
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2							

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list $v_1, v_2, v_3, \ldots$
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

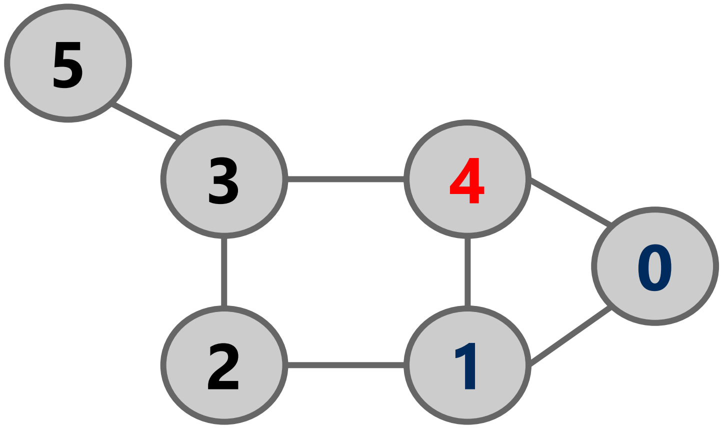
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2	-1	2	1				

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex v, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v)$, $(v_2 - v_1)$, $(v_3 - v_2)$, …



Edges

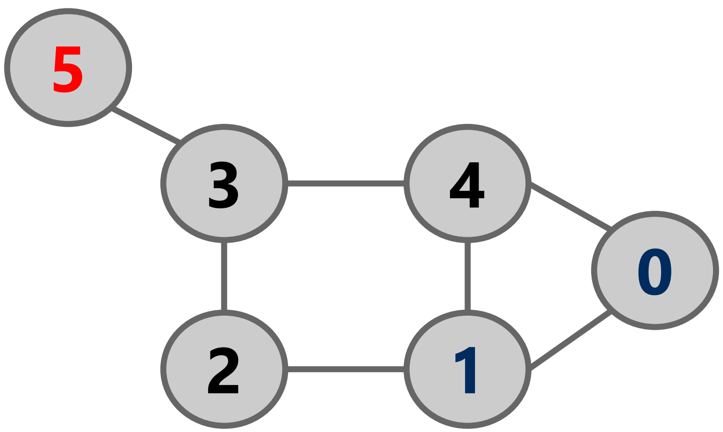
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v)$, $(v_2 - v_1)$, $(v_3 - v_2)$, …



Edges

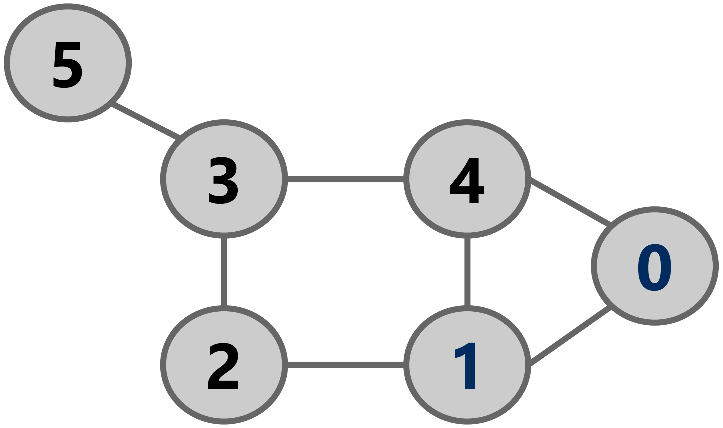
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

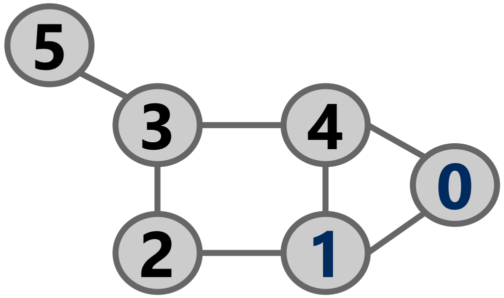
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 3: Use Gamma code to compress the differences**



Edges

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2

Offsets

0	2	5	7	10	13

Gamma Code

- Gamma Code is used to compress data in which small values are much more frequent than large values
- To encode an integer *x*,
  - find T, the largest power of 2 < *x*
  - Encode T as (log T) zeros followed by 1
  - Append the remaining (log T) binary digits of x
- Example: To encode 17: 10001
  - T = 16 = 2^4
  - Gamma code: 0000 1 0001
- 2 floor(log x) + 1
  - much smaller than 32 bits if *x* is small

Graph Compression

- **Goal**: make the differences between vertex labels in each neighbor list small
  - So that their Gamma codes are much less than 32 bits
- For Web Graphs
  - Each vertex is a web page
  - Sort the pages based on their reverse URL (e.g., www.cs.pitt.edu)
  - Most links are local (within the same domain)
    - neighbors will be close to each other in the sorted list
    - Goal achieved
- Other graphs can be relabeled to achieve that goal
  - https://www.cs.cmu.edu/~guyb/papers/BBK03.pdf

Neighborhood connectivity Problem

- We want to keep a set of neighborhoods connected with the minimum cost possible
- **Input:** A set of neighborhoods and a file with the following format:
  - neighborhood i, neighborhood j, cost of connecting the two neighborhoods
  - …
- **Output:** A set of neighborhood pairs to be connected and a total cost such that
  - We can go from any neighborhood to any other **(connected)**
  - The total cost should be minimum (i.e., as small as it can be) **(minimal cost)**

Think Data Structures First!

- How can we structure the input in computer memory?
- Can we use Graphs?
- What about the costs? How can we model that?

We said spatial layouts of graphs were irrelevant

- We define graphs as sets of vertices and edges
- However, we'll certainly want to be able to reason about bandwidth, distance, capacity, etc. of the real world things our graph represents
  - Whether a link is 1 gigabit or 10 megabit will drastically affect our analysis of traffic flowing through a network
  - Having a road between two cities that is a 1 lane country road is very different from having a 4 lane highway
  - If two airports are 2000 miles apart, the number of flights going in and out between them will be drastically different from airports 200 miles apart

We can represent such information with edge weights

- How do we store edge weights?
  - Adjacency matrix?
  - Adjacency list?
  - Do we need a whole new graph representation?
- How do weights affect finding spanning trees/shortest paths?
  - The weighted variants of these problems are called finding the *minimum spanning tree* and the *weighted shortest path*

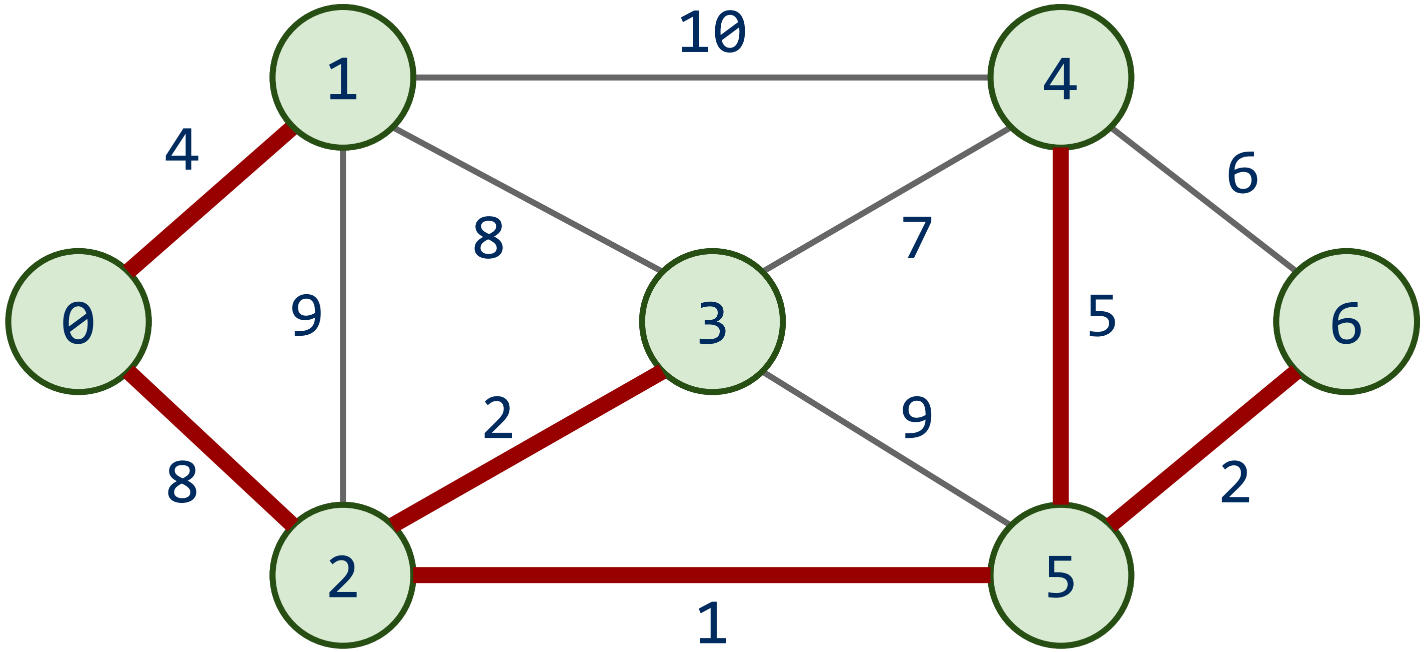
Minimum spanning trees (MST)

- Graphs can potentially have multiple spanning trees
- MST is the spanning tree that has the minimum sum of the weights of its edges

Prim's algorithm

- Initialize T to contain the starting vertex
  - T will eventually become the MST
- While there are vertices not in T:
  - Find minimum edge-weight edge that connects a vertex in T to a vertex not yet in T
  - Add the edge with its vertex to T

Prim's algorithm



Runtime of Prim's

- At each step, check all possible edges
- For a complete graph:
  - First iteration:
    - v - 1 possible edges
  - Next iteration:
    - 2(v - 2) possibilities
      - Each vertex in T shared v-1 edges with other vertices, but the edges they shared with each other already in T
  - Next:
    - 3(v - 3) possibilities
  - ...
- Runtime:
  - $\Sigma_{i\ =\ 1\ \text{to}\ v}\,(i * (v - i)) = \Theta(\text{largest term} * \text{number of terms})$
  - number of terms = *v*
  - largest term is $v^2/4$ (when $i = v/2$)
  - Evaluates to $\Theta(v^3)$

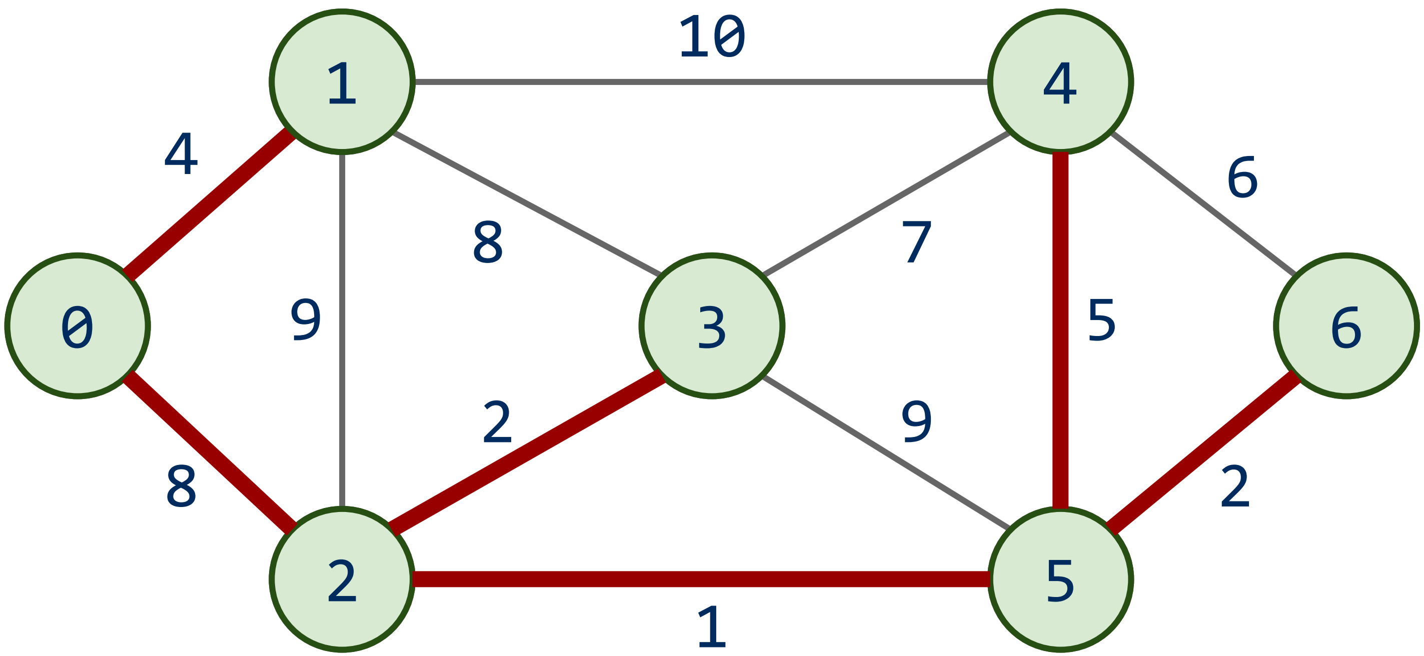
Do we need to look through all remaining edges?

- No! We only need to consider the *best* edge possible for each vertex!
    - The best edge of each vertex can be updated as we add each vertex to T

An enhanced implementation of Prim's Algorithm

- Add start vertex to T
- Search through the neighbors of the added vertex to adjust the parent and best edge arrays as needed
- Search through the best edge array to find the next addition to T
- Repeat until all vertices added to T

Prim's algorithm



	0	1	2	3	4	5	6
Parent:	--	0	0	2	5	2	5
Best Edge:	0	4	8	2	5	1	2

OK, so what's our runtime?

- For every vertex we add to T, we'll need to check all of its neighbors to update their best edges as needed
  - Let's assume we use an **adjacency matrix**:
    - Takes Θ(v) to check the neighbors of a given vertex
    - Time to update parent/best edge arrays?
      - Θ(1)
    - Time to pick next vertex?
      - Θ(v)
    - Total: v*2 Θ(v) = $\Theta(v^2)$

OK, so what's our runtime?

- For every vertex we add to T, we'll need to check all of its neighbors to update their best edges as needed
    - Let's assume we use **adjacency lists**
        - Takes $\Theta(d)$ to check the neighbors of a given vertex
        - Time to update parent/best edge arrays?
            - $\Theta(1)$
        - Time to pick next vertex?
            - $\Theta(v)$
        - Total: $v*\Theta(v + d) = \Theta(v^2)$

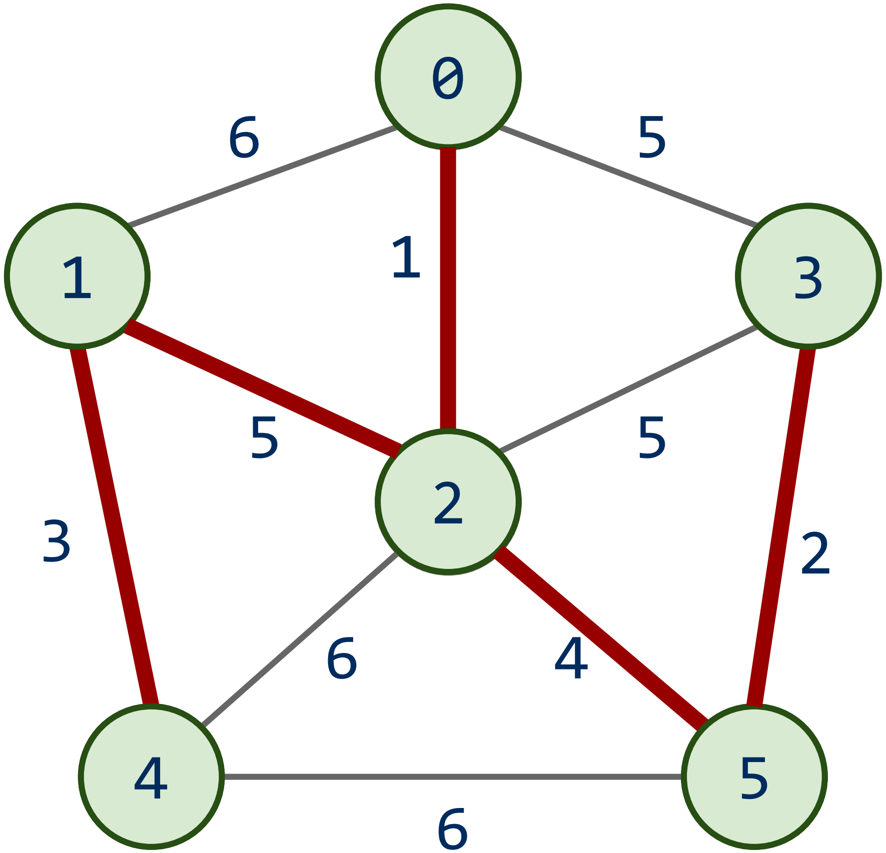
Prim's MST Algorithm

- seen, parent, and BestEdge are arrays of size v
- Initialize seen to false, parent to -1, and BestEdge to infinity
- BestEdge[start] = 0
- for i = 0 to v-1
  - Find a vertex w with seen[w] = false and BestEdge[w] is the minimum over all unseen vertices
  - seen[w] = 1
  - for each neighbor x of w
    - if(BestEdge[x] > edge weight of edge (w, x)
      - BestEdge[x] = edge weight of (w, x)
      - parent[x] = w
- The parent array represents the found MST

What about a faster way to pick the best edge?

- Sounds like a job for a priority queue!
  - Priority queues can remove the min value stored in them in Θ(lg n)
    - Also Θ(lg n) to add to the priority queue
- What does our algorithm look like now?
  - Visit a vertex
  - Add edges coming out of it to a PQ
  - While there are unvisited vertices, pop from the PQ for the next vertex to visit and repeat

Prim's with a priority queue



PQ:

1: (0, 2)

2: (5, 3)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (2, 1)

6: (0, 1)

6: (2, 4)

6: (5, 4)

Runtime using a priority queue

- Have to insert all e edges into the priority queue
  - In the worst case, we'll also have to remove all e edges
- So we have:
  - $e * \Theta(\lg e) + e * \Theta(\lg e)$
  - $= \Theta(2 * e \lg e)$
  - $= \Theta(e \lg e)$
- This algorithm is known as *lazy Prim's*

Do we really need to maintain e items in the PQ?

- I suppose we could not be so lazy
- Just like with the best edge array implementation, we only need the best edge for each vertex
  - PQ will need to be indexable to update the best edge
- This is the idea of *eager Prim's*
  - Runtime is $\Theta(e \lg v)$

Eager Prim's Runtime

v insertions : $v \log v$

e updates : $e \log v$

v removals : $v \log v$

$$(e+v)\log v = \Theta(e \log v)$$

$e \geq (v-1)$

Comparison of Prim's implementations

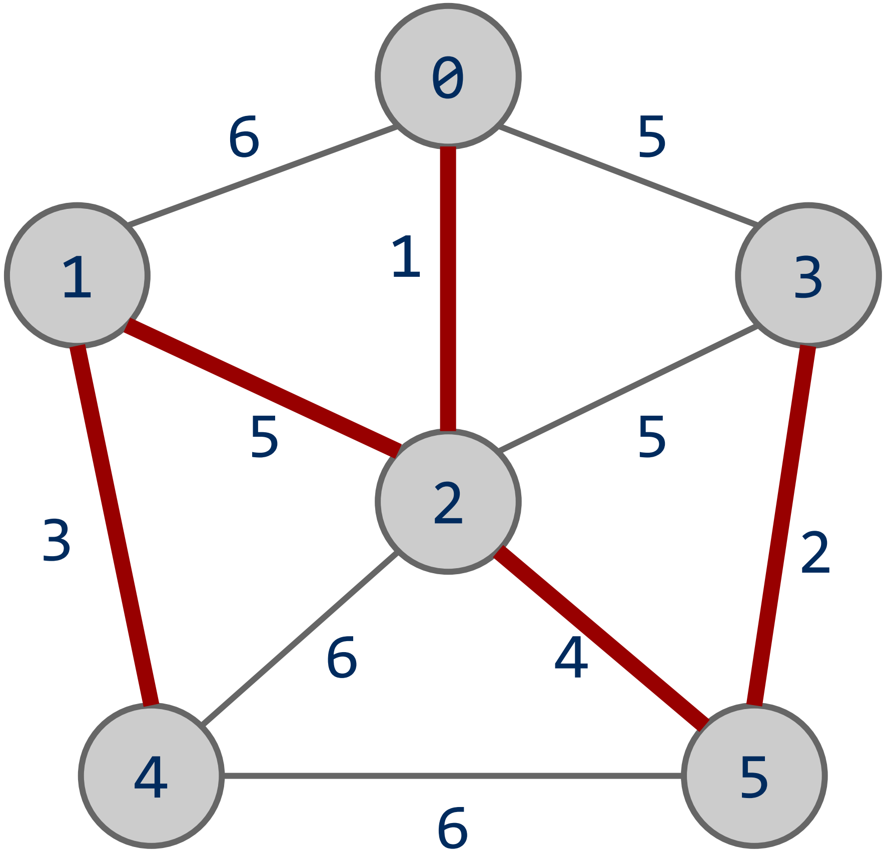
- Parent/Best Edge array Prim's
  - Runtime: $\Theta(v^2)$
  - Space: $\Theta(v)$
- Lazy Prim's
  - Runtime: $\Theta(e \lg e)$
  - Space: $\Theta(e)$
  - Requires a PQ
- Eager Prim's
  - Runtime: $\Theta(e \lg v)$
  - Space: $\Theta(v)$
  - Requires an indexable PQ

How do these compare?

Another MST algorithm

- Kruskal's MST:
  - Insert all edges into a PQ
  - Grab the min edge from the PQ that does not create a cycle in the MST
  - Remove it from the PQ and add it to the MST

Kruskal's example



PQ:

1: (0, 2)

2: (3, 5)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (1, 2)

6: (0, 1)

6: (2, 4)

6: (4, 5)

Kruskal’s runtime

- Instead of building up the MST starting from a single vertex, we build it up using edges all over the graph
- How do we efficiently implement cycle detection?

Kruskal's Runtime

e iterations

remove $\log e$

cycle detection $\Theta(v+e)$

DFS/BFS

$e(v+e) = \Theta(e^2)$